

算法解题笔记

郑华

2017 年 8 月 23 日

目录

第一章 算法基础	27
1.1 时间复杂度	27
1.2 解题提醒	27
1.2.1 等价意思.. 简化题目	27
1.2.2 有无最优子结构	27
1.2.3 想通可行再实践	27
1.3 分治思想	27
1.3.1 基本概念	27
1.3.2 基本思想及策略	28
1.3.3 分治法适用的情况	28
1.3.4 基本步骤	28
1.3.5 可使用分治法求解的一些经典问题	29
1.3.6 依据分治法设计程序时的思维过程	29
1.4 动态规划思想	29
1.4.1 基本概念	29
1.4.2 基本思想和策略	29
1.4.3 适用的情况	30
1.4.4 例题说明	30
1.4.5 题目	31
1.5 贪心算法思想	31
1.5.1 基本概念	31
1.5.2 基本思想和策略	32
1.5.3 适用的情况	32
1.5.4 求解的基本步骤	32
1.5.5 算法实现的说明	32
1.5.6 与动态规划的区别	32
1.6 参考	32
第二章 剑指	33
2.1 基础知识	33
2.1.1 C++	33

2.1.2	数据结构	34
2.1.3	解题分析	35
2.2	高质量代码	35
2.2.1	代码的质量	36
2.2.2	代码的规范性	36
2.2.3	代码的完整性	36
2.2.4	代码的鲁棒性	37
2.2.5	小结	37
2.3	解题思路	38
2.3.1	画图让抽象形象化	38
2.3.2	举例让抽象具体化	38
2.3.3	分解让复杂问题简化	38
2.4	优化时间和空间效率	38
2.4.1	时间效率	38
2.4.2	时间效率与空间效率的平衡	39
2.5	能力体现篇	39
2.5.1	知识迁移能力	39
2.5.2	抽象建模能力	39
2.5.3	思维发散能力	39
第三章	编程之美	41
3.1	数字中的技巧	41
3.1.1	求二进制数中 1 的个数	41
3.1.2	不要被阶乘吓到	41
3.1.3	寻找超过占比 1/2 的 ID	41
3.1.4	1 的数目	42
3.1.5	寻找最大的 k 个数	43
3.1.6	精确表达浮点数	43
3.1.7	最大公约数问题	43
3.1.8	找到符合条件的整数	43
3.1.9	斐波那契数列	43
3.1.10	寻找数组中的最大最小值	43
3.1.11	寻找最近点对	43
3.1.12	找到符合条件的两个数	43
3.1.13	子数组的最大乘积	43
3.1.14	子数组之和的最大值	43
3.1.15	子数组之和的最大值 (二维)	43
3.1.16	求数组中最长递增子序列	43
3.1.17	数组循环移位	44

3.1.18	数组分割	44
3.1.19	区间重合判断	44
3.1.20	程序理解和时间分析	44
3.1.21	只考加法的面试题	44
3.2	字符串及链表的探索	44
3.2.1	字符串移位包含的问题	44
3.2.2	电话号码对应英语单词	44
3.2.3	计算字符串的相似度	44
3.2.4	从无头单链表中删除节点	44
3.2.5	最短摘要的生成	44
3.2.6	判断两个链表是否交叉	44
3.2.7	队列中取最大值操作问题	44
3.2.8	求二叉树中结点的最大距离	44
3.2.9	重建二叉树	44
3.2.10	分层遍历二叉树	47
3.2.11	二分查找	47
3.3	数学游戏	48
3.3.1	金刚坐飞机问题	48
3.3.2	瓷砖覆盖地板	48
3.3.3	买票找零	48
3.3.4	点是否在三角形内	48
3.3.5	磁带文件存放优化	48
3.3.6	桶中取黑白球	48
3.3.7	蚂蚁爬杆	48
3.3.8	三角形测试用例	48
3.3.9	数独知多少	48
3.3.10	数字哑谜和回文	48
3.3.11	扫雷游戏的概率	48
第四章	Array- 80	49
4.1	Two Sum	49
4.1.1	问题	49
4.1.2	解答	49
4.1.3	Normal-两层循环	49
4.1.4	Good-HashTable:HashMap	49
4.2	Median of Two Sorted Arrays	50
4.2.1	分析	51
4.2.2	解答	51
4.3	Word Ladder II	51

4.3.1	分析	51
4.3.2	解答	51
4.4	4Sum	51
4.4.1	分析	52
4.4.2	解答	52
4.5	Rotate Array	52
4.5.1	分析	52
4.5.2	解答	52
4.6	Maximum Product Subarray	52
4.6.1	分析	52
4.6.2	解答	52
4.7	Spiral Matrix	52
4.7.1	分析	53
4.7.2	解答	53
4.8	First Missing Positive	53
4.8.1	分析	53
4.8.2	解答	53
4.9	Word Search	53
4.9.1	分析	54
4.9.2	解答	54
4.10	Largest Rectangle in Histogram	54
4.10.1	分析	54
4.10.2	解答	54
4.11	Jump Game II	54
4.11.1	分析	55
4.11.2	解答	55
4.12	Maximal Rectangle	55
4.12.1	分析	55
4.12.2	解答	55
第五章	Hash Table- 52	57
5.1	Longest Substring Without Repeating Characters	57
5.2		58
5.2.1	分析	58
5.2.2	解答	58
5.3		58
5.3.1	分析	58
5.3.2	解答	58
5.4		58

5.4.1	分析	58
5.4.2	解答	58
5.5		58
5.5.1	分析	58
5.5.2	解答	58
5.6		58
5.6.1	分析	59
5.6.2	解答	59
5.7		59
5.7.1	分析	59
5.7.2	解答	59
5.8		59
5.8.1	分析	59
5.8.2	解答	59
5.9		59
5.9.1	分析	59
5.9.2	解答	59
第六章	Linked List- 27	61
6.1	将单链表反转	61
6.1.1	解答	61
6.2	查找单链表中的倒数第 K 个结点 ($k > 0$)	61
6.2.1	解答	61
6.3	查找单链表的中间结点	61
6.3.1	解答	61
6.4	从尾到头打印单链表	61
6.4.1	分析	61
6.4.2	解答	61
6.5	已知两个单链表 pHead1 和 pHead2 各自有序，把它们合并成一个链表依然有序	61
6.5.1	分析	62
6.5.2	解答	62
6.6	判断一个单链表中是否有环	62
6.6.1	分析	62
6.6.2	解答	62
6.7	判断两个单链表是否相交	62
6.7.1	分析	62
6.7.2	解答	62
6.8	求两个单链表相交的第一个节点	62
6.8.1	分析	62

6.8.2	解答	62
6.9	已知一个单链表中存在环，求进入环中的第一个节点	62
6.9.1	分析	63
6.9.2	解答	63
6.10	给出一单链表头指针 pHead 和一节点指针 pToBeDeleted，O(1) 时间复杂度删除节点 pToBeDeleted	63
6.10.1	分析	63
6.10.2	解答	63
第七章	Math- 58	65
7.1		65
7.1.1	分析	65
7.1.2	解答	65
7.2		65
7.2.1	分析	65
7.2.2	解答	65
7.3		65
7.3.1	分析	65
7.3.2	解答	65
7.4		65
7.4.1	分析	66
7.4.2	解答	66
7.5		66
7.5.1	分析	66
7.5.2	解答	66
7.6		66
7.6.1	分析	66
7.6.2	解答	66
7.7		66
7.7.1	分析	66
7.7.2	解答	66
7.8		66
7.8.1	分析	66
7.8.2	解答	66
7.9		66
7.9.1	分析	67
7.9.2	解答	67
7.10		67
7.10.1	分析	67

7.10.2	解答	67
第八章	Two Pointers- 33	69
8.1		69
8.1.1	分析	69
8.1.2	解答	69
8.2		69
8.2.1	分析	69
8.2.2	解答	69
8.3		69
8.3.1	分析	69
8.3.2	解答	69
8.4		69
8.4.1	分析	70
8.4.2	解答	70
8.5		70
8.5.1	分析	70
8.5.2	解答	70
8.6		70
8.6.1	分析	70
8.6.2	解答	70
8.7		70
8.7.1	分析	70
8.7.2	解答	70
8.8		70
8.8.1	分析	70
8.8.2	解答	70
8.9		70
8.9.1	分析	71
8.9.2	解答	71
8.10		71
8.10.1	分析	71
8.10.2	解答	71
第九章	String- 55	73
9.1		73
9.1.1	分析	73
9.1.2	解答	73
9.2		73
9.2.1	分析	73

9.2.2	解答	73
9.3		73
9.3.1	分析	73
9.3.2	解答	73
9.4		73
9.4.1	分析	74
9.4.2	解答	74
9.5		74
9.5.1	分析	74
9.5.2	解答	74
9.6		74
9.6.1	分析	74
9.6.2	解答	74
9.7		74
9.7.1	分析	74
9.7.2	解答	74
9.8		74
9.8.1	分析	74
9.8.2	解答	74
9.9		74
9.9.1	分析	75
9.9.2	解答	75
9.10		75
9.10.1	分析	75
9.10.2	解答	75
9.11		75
9.11.1	分析	75
9.11.2	解答	75
9.12		75
9.12.1	分析	75
9.12.2	解答	75
9.13		75
9.13.1	分析	75
9.13.2	解答	75
9.14		75
9.14.1	分析	76
9.14.2	解答	76
9.15		76
9.15.1	分析	76

9.15.2 解答	76
9.16 Divide and Conquer- 12	76
9.17	76
9.17.1 分析	76
9.17.2 解答	76
9.18	76
9.18.1 分析	76
9.18.2 解答	76
9.19	76
9.19.1 分析	76
9.19.2 解答	76
9.20	76
9.20.1 分析	77
9.20.2 解答	77
9.21	77
9.21.1 分析	77
9.21.2 解答	77
9.22	77
9.22.1 分析	77
9.22.2 解答	77
9.23	77
9.23.1 分析	77
9.23.2 解答	77
9.24	77
9.24.1 分析	77
9.24.2 解答	77
9.25	77
9.25.1 分析	78
9.25.2 解答	78
9.26	78
9.26.1 分析	78
9.26.2 解答	78
第十章 Binary Search- 38	79
10.1	79
10.1.1 分析	79
10.1.2 解答	79
10.2	79
10.2.1 分析	79

10.2.2	解答	79
10.3		79
10.3.1	分析	79
10.3.2	解答	79
10.4		79
10.4.1	分析	80
10.4.2	解答	80
10.5		80
10.5.1	分析	80
10.5.2	解答	80
10.6		80
10.6.1	分析	80
10.6.2	解答	80
10.7		80
10.7.1	分析	80
10.7.2	解答	80
10.8		80
10.8.1	分析	80
10.8.2	解答	80
10.9		80
10.9.1	分析	81
10.9.2	解答	81
10.10		81
10.10.1	分析	81
10.10.2	解答	81

第十一章 Dynamic Programming- 66 83

11.1		83
11.1.1	分析	83
11.1.2	解答	83
11.2		83
11.2.1	分析	83
11.2.2	解答	83
11.3		83
11.3.1	分析	83
11.3.2	解答	83
11.4		83
11.4.1	分析	84
11.4.2	解答	84

11.5	84
11.5.1 分析	84
11.5.2 解答	84
11.6	84
11.6.1 分析	84
11.6.2 解答	84
11.7	84
11.7.1 分析	84
11.7.2 解答	84
11.8	84
11.8.1 分析	84
11.8.2 解答	84
11.9	84
11.9.1 分析	85
11.9.2 解答	85
11.10	85
11.10.1 分析	85
11.10.2 解答	85
11.11	85
11.11.1 分析	85
11.11.2 解答	85
11.12	85
11.12.1 分析	85
11.12.2 解答	85
11.13	85
11.13.1 分析	85
11.13.2 解答	85
11.14	85
11.14.1 分析	86
11.14.2 解答	86
11.15	86
11.15.1 分析	86
11.15.2 解答	86
11.16	86
11.16.1 分析	86
11.16.2 解答	86
11.17	86
11.17.1 分析	86
11.17.2 解答	86

11.18	86
11.18.1 分析	86
11.18.2 解答	86
第十二章 Backtracking- 34	87
12.1	87
12.1.1 分析	87
12.1.2 解答	87
12.2	87
12.2.1 分析	87
12.2.2 解答	87
12.3	87
12.3.1 分析	87
12.3.2 解答	87
12.4	87
12.4.1 分析	88
12.4.2 解答	88
12.5	88
12.5.1 分析	88
12.5.2 解答	88
12.6	88
12.6.1 分析	88
12.6.2 解答	88
12.7	88
12.7.1 分析	88
12.7.2 解答	88
第十三章 Stack- 25	89
13.1	89
13.1.1 分析	89
13.1.2 解答	89
13.2	89
13.2.1 分析	89
13.2.2 解答	89
13.3	89
13.3.1 分析	89
13.3.2 解答	89
13.4	89
13.4.1 分析	90
13.4.2 解答	90

13.5	90
13.5.1 分析	90
13.5.2 解答	90
第十四章 Heap- 15	91
14.1	91
14.1.1 分析	91
14.1.2 解答	91
14.2	91
14.2.1 分析	91
14.2.2 解答	91
14.3	91
14.3.1 分析	91
14.3.2 解答	91
14.4	91
14.4.1 分析	92
14.4.2 解答	92
14.5	92
14.5.1 分析	92
14.5.2 解答	92
14.6	92
14.6.1 分析	92
14.6.2 解答	92
14.7	92
14.7.1 分析	92
14.7.2 解答	92
14.8	92
14.8.1 分析	92
14.8.2 解答	92
14.9	92
14.9.1 分析	93
14.9.2 解答	93
14.10	93
14.10.1 分析	93
14.10.2 解答	93
第十五章 Greedy- 19	95
15.1	95
15.1.1 分析	95
15.1.2 解答	95

15.2	95
15.2.1 分析	95
15.2.2 解答	95
15.3	95
15.3.1 分析	95
15.3.2 解答	95
15.4	95
15.4.1 分析	96
15.4.2 解答	96
15.5	96
15.5.1 分析	96
15.5.2 解答	96
15.6	96
15.6.1 分析	96
15.6.2 解答	96
15.7	96
15.7.1 分析	96
15.7.2 解答	96
15.8	96
15.8.1 分析	96
15.8.2 解答	96
15.9	96
15.9.1 分析	97
15.9.2 解答	97
15.10	97
15.10.1 分析	97
15.10.2 解答	97

第十六章 Sort- 16 99

16.1 冒泡	99
16.1.1 解答	99
16.2 选择排序	100
16.2.1 解答	100
16.3 快速排序	100
16.3.1 解答	100
16.4 插入排序	100
16.4.1 解答	100
16.5 归并	100
16.5.1 解答	101

16.6 堆	101
16.6.1 解答	101
16.7	101
16.7.1 分析	102
16.7.2 解答	102
16.8	102
16.8.1 分析	102
16.8.2 解答	102
16.9	102
16.9.1 分析	102
16.9.2 解答	102
16.10	102
16.10.1 分析	102
16.10.2 解答	102
第十七章 Bit Manipulation- 26	103
17.1	103
17.1.1 分析	103
17.1.2 解答	103
17.2	103
17.2.1 分析	103
17.2.2 解答	103
17.3	103
17.3.1 分析	103
17.3.2 解答	103
17.4	103
17.4.1 分析	104
17.4.2 解答	104
17.5	104
17.5.1 分析	104
17.5.2 解答	104
17.6	104
17.6.1 分析	104
17.6.2 解答	104
17.7	104
17.7.1 分析	104
17.7.2 解答	104
17.8	104
17.8.1 分析	104

17.8.2 解答	104
17.9	104
17.9.1 分析	105
17.9.2 解答	105
17.10	105
17.10.1 分析	105
17.10.2 解答	105
第十八章 Tree- 48	107
18.1 Binary Tree Maximum Path Sum	107
18.1.1 分析	107
18.1.2 解答	107
18.2 Validate Binary Search Tree	108
18.2.1 分析	110
18.2.2 解答	110
18.3 求二叉树中的节点个数	110
18.3.1 分析	110
18.3.2 解答	110
18.4 求二叉树的深度	110
18.4.1 分析	110
18.4.2 解答	110
18.5 前序遍历，中序遍历，后序遍历	110
18.5.1 分析	110
18.5.2 解答	110
18.6 分层遍历二叉树（按层次从上往下，从左往右）	110
18.6.1 分析	110
18.6.2 解答	110
18.7 将二叉查找树变为有序的双向链表	110
18.7.1 分析	110
18.7.2 解答	110
18.8 求二叉树第 K 层的节点个数	110
18.8.1 分析	110
18.8.2 解答	110
18.9 求二叉树中叶子节点的个数	110
18.9.1 分析	110
18.9.2 解答	110
18.10 判断两棵二叉树是否结构相同	110
18.10.1 分析	110
18.10.2 解答	110

18.11 判断二叉树是不是平衡二叉树	110
18.11.1 分析	110
18.11.2 解答	110
18.12 求二叉树的镜像	110
18.12.1 分析	110
18.12.2 解答	110
18.13 求二叉树中两个节点的最低公共祖先节点	110
18.13.1 分析	110
18.13.2 解答	110
18.14 求二叉树中节点的最大距离	110
18.14.1 分析	110
18.14.2 解答	110
18.15 由前序遍历序列和中序遍历序列重建二叉树	110
18.15.1 分析	110
18.15.2 解答	110
18.16 判断二叉树是不是完全二叉树	110
18.16.1 分析	110
18.16.2 解答	110

第十九章 Depth-first Search- 41 111

19.1 部分和问题	111
19.1.1 分析	111
19.1.2 解答	111
19.2	112
19.2.1 分析	112
19.2.2 解答	112
19.3	112
19.3.1 分析	112
19.3.2 解答	112
19.4	112
19.4.1 分析	112
19.4.2 解答	112
19.5	112
19.5.1 分析	112
19.5.2 解答	112
19.6	112
19.6.1 分析	113
19.6.2 解答	113
19.7	113

19.7.1	分析	113
19.7.2	解答	113
19.8		113
19.8.1	分析	113
19.8.2	解答	113
19.9		113
19.9.1	分析	113
19.9.2	解答	113
19.10		113
19.10.1	分析	113
19.10.2	解答	113

第二十章 Breadth-first Search- 21 115

20.1		115
20.1.1	分析	115
20.1.2	解答	115
20.2		115
20.2.1	分析	115
20.2.2	解答	115
20.3		115
20.3.1	分析	115
20.3.2	解答	115
20.4		115
20.4.1	分析	116
20.4.2	解答	116
20.5		116
20.5.1	分析	116
20.5.2	解答	116
20.6		116
20.6.1	分析	116
20.6.2	解答	116
20.7		116
20.7.1	分析	116
20.7.2	解答	116
20.8		116
20.8.1	分析	116
20.8.2	解答	116
20.9		116
20.9.1	分析	117

20.9.2 解答	117
20.10	117
20.10.1 分析	117
20.10.2 解答	117
第二十一章 Union Find- 6	119
21.1	119
21.1.1 分析	119
21.1.2 解答	119
21.2	119
21.2.1 分析	119
21.2.2 解答	119
21.3	119
21.3.1 分析	119
21.3.2 解答	119
21.4	119
21.4.1 分析	119
21.4.2 解答	119
第二十二章 Graph- 10	121
22.1	121
22.1.1 分析	121
22.1.2 解答	121
22.2	121
22.2.1 分析	121
22.2.2 解答	121
22.3	121
22.3.1 分析	121
22.3.2 解答	121
22.4	121
22.4.1 分析	122
22.4.2 解答	122
22.5	122
22.5.1 分析	122
22.5.2 解答	122
22.6	122
22.6.1 分析	122
22.6.2 解答	122
22.7	122
22.7.1 分析	122

22.7.2 解答	122
22.8	122
22.8.1 分析	122
22.8.2 解答	122
22.9	122
22.9.1 分析	123
22.9.2 解答	123
22.10	123
22.10.1 分析	123
22.10.2 解答	123

第二十三章 Design- 27 125

23.1	125
23.1.1 分析	125
23.1.2 解答	125
23.2	125
23.2.1 分析	125
23.2.2 解答	125
23.3	125
23.3.1 分析	125
23.3.2 解答	125
23.4	125
23.4.1 分析	126
23.4.2 解答	126
23.5	126
23.5.1 分析	126
23.5.2 解答	126
23.6	126
23.6.1 分析	126
23.6.2 解答	126
23.7	126
23.7.1 分析	126
23.7.2 解答	126
23.8	126
23.8.1 分析	126
23.8.2 解答	126
23.9	126
23.9.1 分析	127
23.9.2 解答	127

23.10	127
23.10.1 分析	127
23.10.2 解答	127
第二十四章 Topological Sort[拓扑排序]- 5	129
24.1	129
24.1.1 分析	129
24.1.2 解答	129
24.2	129
24.2.1 分析	129
24.2.2 解答	129
24.3	129
24.3.1 分析	129
24.3.2 解答	129
第二十五章 Trie[前缀-字典树]- 7	131
25.1	131
25.1.1 分析	131
25.1.2 解答	131
25.2	131
25.2.1 分析	131
25.2.2 解答	131
25.3	131
25.3.1 分析	131
25.3.2 解答	131
第二十六章 Binary Indexed Tree- 4	133
26.1	133
26.1.1 分析	133
26.1.2 解答	133
26.2	133
26.2.1 分析	133
26.2.2 解答	133
26.3	133
26.3.1 分析	133
26.3.2 解答	133
第二十七章 Segment Tree- 4	135
27.1	135
27.1.1 分析	135
27.1.2 解答	135

27.2	135
27.2.1 分析	135
27.2.2 解答	135
27.3	135
27.3.1 分析	135
27.3.2 解答	135
第二十八章 Binary Search Tree- 4	137
28.1	137
28.1.1 分析	137
28.1.2 解答	137
28.2	137
28.2.1 分析	137
28.2.2 解答	137
28.3	137
28.3.1 分析	137
28.3.2 解答	137
第二十九章 Other Aspects	139
29.1 Recursion[递归]- 2	139
29.1.1	139
29.1.2 分析	139
29.1.3 解答	139
29.2 Brainteaser[谜题]- 2	139
29.2.1	139
29.2.2 分析	139
29.2.3 解答	139
29.3 Memoization[以避免递归重复计算，如 Fibonacci(斐波那契) 问题]- 1	139
29.3.1	139
29.3.2 分析	140
29.3.3 解答	140
29.4 Queue- 3	140
29.4.1	140
29.4.2	140
29.5 Reservoir Sampling[随机抽样]- 2	140
29.5.1	140
29.6 Minimax[极小化极大算法]- 3	140
29.6.1	140
29.6.2	140
29.7 Other Interview	141

29.7.1 迷宫字符串	141
------------------------	-----

第一章 算法基础

1.1 时间复杂度

n	\log_2^n	$n \times \log_2^n$	n^2	n^3	2^n	$n!$
4	2	8	16	64	16	24
8	3	24	64	512	256	80320
10	3.32	33.2	100	1000	1024	3628800
16	4	64	256	4096	65536	2.1×10^{13}
32	5	160	1024	32768	4.3×10^9	2.6×10^{35}
128	7	896	16384	2097152	3.4×10^{38}	∞
1024	10	10240	1048576	1.07×10^9	∞	∞
10000	13.29	132877	10^8	10^{12}	∞	∞

表 1.1: 各个函数随 n 的增长函数值的变化

1.2 解题提醒

1.2.1 等价意思.. 简化题目

1.2.2 有无最优子结构

即是否可以用数学公式简化模型。

1.2.3 想通可行再实践

1.3 分治思想

1.3.1 基本概念

在计算机科学中，分治法是一种很重要的算法。字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。这个技巧是很多高效算法的基础，如排序算法 (快速排序，归并排序)，傅立叶变换 (快速傅立叶变换)……

任何一个可以用计算机求解的问题所需的计算时间都与其规模有关。问题的规模越小，越容易直接求解，解题所需的计算时间也越少。例如，对于 n 个元素的排序问题，当 $n=1$ 时，不需任何计算。 $n=2$ 时，只要作一次比较即可排好序。 $n=3$ 时只要作 3 次比较即可，…。而当 n 较大时，问题就不那么容易处理了。要想直接解决一个规模较大的问题，有时是相当困难的。

1.3.2 基本思想及策略

分治法的设计思想是：将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

分治策略是：对于一个规模为 n 的问题，若该问题可以容易地解决（比如说规模 n 较小）则直接解决，否则将其分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。

如果原问题可分割成 k 个子问题， $1 < k \leq n$ ，且这些子问题都可解并可利用这些子问题的解求出原问题的解，那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

1.3.3 分治法适用的情况

分治法所能解决的问题一般具有以下几个特征：

- 该问题的规模缩小到一定的程度就可以容易地解决：绝大多数问题都可以满足的，因为问题的计算复杂性一般是随着问题规模的增加而增加；
- 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质：应用分治法的前提它也是大多数问题可以满足的，此特征反映了递归思想的应用；
- 利用该问题分解出的子问题的解可以合并为该问题的解：是关键，能否利用分治法完全取决于问题是否具有第三条特征，如果具备了第一条和第二条特征，而不具备第三条特征，则可以考虑用贪心法或动态规划法。
- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题：涉及到分治法的效率，如果各子问题是不独立的则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然可用分治法，但一般用动态规划法较好

1.3.4 基本步骤

分治法在每一层递归上都有三个步骤：

- step1 分解：**将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；
- step2 解决：**若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题
- step3 合并：**将各个子问题的解合并为原问题的解。

1.3.5 可使用分治法求解的一些经典问题

- 二分搜索
- 大整数乘法
- Strassen 矩阵乘法
- 棋盘覆盖
- 合并排序
- 快速排序
- 线性时间选择
- 最接近点对问题
- 循环赛日程表
- 汉诺塔

1.3.6 依据分治法设计程序时的思维过程

- 1、一定是先找到最小问题规模时的求解方法
- 2、然后考虑随着问题规模增大时的求解方法
- 3、找到求解的递归函数式后（各种规模或因子），设计递归程序即可。

1.4 动态规划思想

<http://mp.weixin.qq.com/s/3h9iqU4rdH3EIy5m6AzXsg>

1.4.1 基本概念

动态规划过程是：每次决策依赖于当前状态，又随即引起状态的转移。一个决策序列就是在变化的状态中产生出来的，所以，这种多阶段最优化决策解决问题的过程就称为动态规划。

1.4.2 基本思想和策略

基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

由于动态规划解决的问题多数有重叠子问题这个特点，为减少重复计算，对每一个子问题只解一次，将其不同阶段的不同状态保存在一个二维数组中。

与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

1.4.3 适用的情况

能采用动态规划求解的问题的一般要具有 3 个性质：

(1) **最优化原理**：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。

(2) **无后效性**：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。

(3) **有重叠子问题**：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）

1.4.4 例题说明

上楼梯

有一座高度是 10 级台阶的楼梯，从下往上走，每跨一步只能向上 1 级或者 2 级台阶。要求用程序来求出一共有多少种走法。

比如，每次走 1 级台阶，一共走 10 步，这是其中一种走法。我们可以简写成 1,1,1,1,1,1,1,1,1,1。

再比如，每次走 2 级台阶，一共走 5 步，这是另一种走法。我们可以简写成 2,2,2,2,2。

当然，除此之外，还有很多很多种走法。

穷举法 二叉树

动态规划法

分析 设走到 m 层需要 x 种走法，这个套路大家肯定非常熟悉，中学数学最喜欢设各种东西。好，现在要开始求解 x 了，首先我们自然会想到先找找看这个 x 跟什么有关系，看能不能跟谁联立一个方程啥的不就好整了么？

问题本身会提供有用信息，这是中学数学的套路，我们看到题目说每一次只能跨一步或两步，也就是说要想到达 m 层，我们只能从 $m-1$ 层跨一步上去或者从 $m-2$ 层跨两步上去。假设到达 $m-1$ 层有 x_1 种走法，到达 $m-2$ 层有 x_2 种走法，那么：

$$x = x_1 + x_2$$

这块有个思维陷阱，就是很容易想到从 $m-2$ 层到 m 层有两种方法，所以 $x = x_1 + 2 * x_2$ ，这样思考是对的，是包含了一步步上的和 2 步上的，但是忽略了 x_1 包含了第一种情况，故 x_2 仅剩下 2 步到达 m 的情况了。

要点从上式中可以看出所求到达 m 层的走法 x 是依附于到达 $m-1$ 层的走法 x_1 和到达 $m-2$ 层的走法 x_2 。看到没有，一个问题的解依附于其子问题的解 (动态规划的精髓所在)。也就是说只要知道了 $m-1$ 层的走法 x_1 和 $m-2$ 层的走法 x_2 就能知道到达 m 层的走法 x 了。

既然我们知道 m 层的走法能从 $m-1$ 层和 $m-2$ 层的走法求得，那同样的道理， $m-1$ 层的走法不也可以从 $(m-1)-1$ 和 $(m-1)-2$ 层的走法求得，换种简单的描述方法，用 $F(m)$ 表示到达 m 层的方法数，那么就有

$$F(m) = F(m-1) + F(m-2)$$

$$F(m-1) = F(m-1-1) + F(m-1-2)$$

当只有 1 个台阶和 2 个台阶时，显然可以看出分别有 1 种与 2 种走法

$$F(1) = 1$$

$$F(2) = 2$$

理论对应 动态规划包括 3 个重要的概念

- **最优子结构**：我们分析到的 $x = x_1 + x_2$ 就是最优子结构，将主问题的解依附于相似的子问题的假设解上。
- **状态转移公式**： $F(m) = F(m-1) + F(m-2)$ 一个问题的解依附于其子问题的解
- **边界**：当只有 1 级和 2 级台阶时，我们可以直接得到结果，无需继续简化，这时我们称 $F(1)$ 与 $F(2)$ 为问题的边界

求解技巧 存储已经计算过的结果，避免重复计算。

1.4.5 题目

入门<http://www.cnblogs.com/zadosu/p/6556000.html>

进阶<http://www.cnblogs.com/zadosu/p/6556392.html>

1.5 贪心算法思想

1.5.1 基本概念

所谓贪心算法是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的仅是在某种意义上的局部最优解。

贪心算法没有固定的算法框架，算法设计的关键是贪心策略的选择。必须注意的是，贪心算法不是对所有问题都能得到整体最优解，选择的贪心策略必须具备无后效性，即某个状态以后的过程不会影响以前的状态，只与当前状态有关。

所以对所采用的贪心策略一定要仔细分析其是否满足无后效性。

1.5.2 基本思想和策略

- 1、建立数学模型来描述问题。
- 2、把求解的问题分成若干个子问题。
- 3、对每一子问题求解，得到子问题的局部最优解。
- 4、把子问题的解局部最优解合成原来解问题的一个解。

1.5.3 适用的情况

贪心策略适用的前提是：局部最优策略能导致产生全局最优解。

实际上，贪心算法适用的情况很少。一般，对一个问题分析是否适用于贪心算法，可以先选择该问题下的几个实际数据进行分析，就可做出判断。

1.5.4 求解的基本步骤

从问题的某一初始解出发；

```
while(能朝给定总目标前进一步)
{
    利用可行的决策，求出可行解的一个解元素；
}
```

由所有解元素组合成问题的一个可行解；

1.5.5 算法实现的说明

背包问题

1.5.6 与动态规划的区别

<http://blog.csdn.net/jarvischu/article/details/6056387>

1.6 参考

算法过来人 + 面试题:<http://blog.csdn.net/qingyuanluofeng/article/details/47612589>

第二章 剑指

2.1 基础知识

2.1.1 C++

面试题 1：赋值运算符 核心考点：

1. 返回为引用，否则无法连续赋值
2. 参数声明为常引用
3. 忘记分配新内存之前释放原有内存会内存泄露
4. 判断参数与当前实例是否相同

```
class CMyString1
{
public:
    CMyString1(char* pData = NULL);
    CMyString1(const CMyString1& str);
    ~CMyString1(void);

    //通过创建临时对象，再让临时对象与原有对象交换，临时对象指向原来的内存，出括号后释放
    CMyString1& operator = (const CMyString1& str1)
    {
        if(this != &str1)
        {
            CMyString strTemp(str1); //在构造函数中用new分配内存，若有异常，状态有效
            char* tmpP = strTemp.m_pData;
            strTemp.m_pData = m_pData;
            m_pData = tmpP;
        }
        return *this;
    }
private:
    char* m_pData;
};
```

面试题 2 实现 Singleton 模式 核心考点:

1. singleton 模型实现
2. 线程安全如何保证- doubleCheck
3. private 构造
4. static 成员

```
template<typename T>
class Singleton
{
public:
    static T& getInstance()
    {
        if(!value_)
        {
            MutexGuard guard(mutex_);
            if (!value_)
            {
                value_ = new T();
            }
        }
        return *value_;
    }

private:
    Singleton();
    ~Singleton();

    static T* value_;
    static Mutex mutex_;
};

template<typename T>
T* Singleton<T>::value_ = NULL;

template<typename T>
Mutex Singleton<T>::mutex_;
```

面试题 48 不能被继承的类

2.1.2 数据结构

线性表

面试题 5 从尾到头打印链表

面试题 7 用两个栈实现队列 ->

```
void EnQueue(Stack *s1, Stack *s2, int k)
{
    Push(s1, k);
}

int DeQueue(Stack *s1, Stack*s2)
{
    if(IsStackEmpty(s2) == 1)
    {
        while(IsStackEmpty(s1) == 0)
        {
            Push(s2, Pop(s1));
        }
    }
    if(IsStackEmpty(s2) == 1)
    {
        printf("Empty!\n");
    }
    return Pop(s2);
}
```

面试题 8 旋转数组的最小数字

树

面试题 6 二叉树中前序确定后续 参考<http://www.cnblogs.com/edisonchou/p/4741099.html>

堆

红黑树

B 树

hash table

2.1.3 解题分析

面试题 10 二进制中 1 的个数

查找数组中第 k 大的数字

2.2 高质量代码

<http://blog.csdn.net/oMengLiShuiXiang1234/article/details/51785436>

2.2.1 代码的质量

下面是几个面试官对代码质量的要求

- 代码的容错能力，对一些特别的输入需要考虑异常状况，考虑资源的回收问题。
- 一些基本的知识点，如 double 类型的数据比较的问题。if(d1==d2) 上述比较有问题，由于精度原因不能用等号判断两个小数是否相等。
- 不能忽略边界的情况
- 变量、函数命名的问题，而且解决一个具体的问题，需要有个合适的数据结构。
- 从程序的正确性和鲁棒性两方面检验代码的质量。关注对输入参数的检查、处理错误和异常的方式、命名方式等

2.2.2 代码的规范性

面试官是根据应聘者写出的代码来决定是否录用他的。如果应聘者代码写的不够规范，影响面试官阅读代码的兴致，那面试官就会默默地减去几分。

- 规范的代码书写清晰

写代码前形成清晰的思路并能把思路用编程语言清楚地书写出来。

- 规范的代码布局清晰

当循环、判断较多，逻辑较复杂时，缩进的层次可能会比较多。

- 规范的代码命名合理

建议：在写代码的时候，用完整的英文单词组合命名变量和函数。

2.2.3 代码的完整性

在面试的过程中，面试官会非常关注应聘者考虑问题是否周全。面试官通过检查代码是否完整来考查应聘者的思维是否全面。通常面试官会检查应聘者的代码是否完成了基本功能、输入边界值是否能得到正确的输出、是否对各种不合规范的非法输入做出了合理的错误处理。

面试题 11 数值的整数次方

面试题 12 打印 1 到最大的 n 位数

面试题 13 在 $O(1)$ 时间删除链表节点

面试题 14 调整数组顺序使奇数位于偶数前面

2.2.4 代码的鲁棒性

鲁棒性 (Robust)，有时也翻译成健壮性。所谓鲁棒性是指程序能够判断输入是否合乎规范要求，并对不合要求的输入予以合理的处理。

容错性是鲁棒性的一个重要体现。不鲁棒性的软件在发生异常事件的时候，比如用户输入错误的用户名、试图打开的文件不存在或者网络不能连接，就会出现不可预见的诡异行为，或者干脆整个软件崩溃。这样的软件对于用户而言，不亚于一场灾难。

由于鲁棒性对软件开发非常重要，面试官在招聘的时候对应聘者写出的代码是否具有鲁棒性也非常关注。提高代码的鲁棒性的有效途径是进行防御性编程。防御性编程是一种编程习惯，是指遇见在什么地方可能会出现问题，并为这些可能出现的问题制定处理方式。比如试图打开文件时发现文件不存在，我们可以提示用户检查文件名和路径；当服务器连接不上时，我们可以试图连接备用服务器等。这样异常发生时，软件的行为也尽在我们的掌握之中，而不至于出现不可预见的事情。

在面试时，最简单也是最使用的防御性编程就是在函数入口添加代码以验证用户输入是否符合要求。通常面试要求写一两个函数，我们需要格外关注这些函数的输入参数。如果输入的是一个指针，那指针时空指针怎么办？如果输入的是一个字符串，那么字符串的内容为空怎么办？如果能把这些问题都提前考虑到，并做相应的处理，那么面试官就会觉得我们有防御性编程的习惯，能够写出鲁棒性的软件。

当然并不是所有与鲁棒性相关的问题都只是检查输入的参数这么简单。我们看到问题的时候，要多问几个“如果不... 那么...”这样的问题。比如面试题 15 “链表中倒数第 K 个结点”，这里隐含一个条件就是链表中结点的个数大于 k。我们要问如果链表中的结点的数目不是大于 k 个，那么，代码会出现什么问题？主要的思考方式能够帮助我们发现潜在的问题并提前解决问题。这比让面试官发现问题之后我们再去慌忙分析代码查找问题的根源要好得多。

面试题 15 链表中倒数第 k 个结点

面试题 16 反转链表

面试题 17 合并两个排序的链表

面试题 18 树的子结构

2.2.5 小结

- 规范性：书写清晰、布局清晰、命名合理
- 完整性：完成基本功能、考虑边界条件、做好错误处理
- 鲁棒性：采取防御式编程、处理无效的输入

2.3 解题思路

2.3.1 画图让抽象形象化

面试题 19 二叉树的镜像

面试题 20 顺时针打印矩阵

2.3.2 举例让抽象具体化

面试题 21 包含 min 函数的栈

面试题 22 栈的压入、弹出序列

面试题 23 从上往下打印二叉树

面试题 24 二叉搜索树的后序遍历序列

面试题 25 二叉树中和为某一值的路径

2.3.3 分解让复杂问题简化

面试题 26 复杂链表的复制

面试题 27 二叉搜索树与双向链表

面试题 28 字符串的排列

2.4 优化时间和空间效率

2.4.1 时间效率

面试题 29 数组中出现次数超过一半的数字

面试题 30 最小的 k 个数

面试题 31 连续子数组的最大和

面试题 32 从 1 到 n 整数中 1 出现的次数

面试题 33 把数组排成最小的数

2.4.2 时间效率与空间效率的平衡

面试题 34 丑数

面试题 35 第一个只出现一次的字符

面试题 36 数组中的逆序对

面试题 37 两个链表的第一个公共结点

2.5 能力体现篇

2.5.1 知识迁移能力

面试题 38 数字在排序数组中出现的次数

面试题 39 二叉树的深度

面试题 40 数组中只出现一次的数字

面试题 41 和为 s 的两个数字

面试题 42 反转单词顺序

2.5.2 抽象建模能力

面试题 43 n 个骰子的点数

面试题 44 扑克牌的顺子

面试题 45 圆圈中最后剩下的数字

2.5.3 思维发散能力

面试题 46 求 $1+2+\dots+n$

面试题 47 不用加减乘除做加法

面试题 49 把字符串转换成整数

面试题 50 树中两个节点的最低公共祖先

第三章 编程之美

<http://blog.csdn.net/qingyuanluofeng/article/category/2544589>

3.1 数字中的技巧

3.1.1 求二进制数中 1 的个数

对于一个字节的无符号整型变量，求其二进制表示中 1 的个数，要求算法的执行效率尽可能高。

3.1.2 不要被阶乘吓到

3.1.3 寻找超过占比 1/2 的 ID

本质: 寻找出现超过一半次数的人

- 转化为更小的问题-Delete Copied
- 如何避免排序超找重复

```
#include<iostream>
#include<vector>
using namespace std;

int find1(vector<int>&id)
{
    int a;
    int ta=0;

    int len = id.size();

    for (int i = 0; i < len; i++){
        if (ta == 0){
            a = id[i];
            ta = 1;
        }
        else{
```

```

        if (a == id[i])
            ta++;
        else
            ta--;
    }
}

return a;
}

```

3.1.4 1 的数目

给定一个十进制正整数 N，写下从 1 开始，到 N 的所有整数然后数一下其中出现的所有 1 的个数 例如：

N=2, 写下 1,2 这样只出现了 1 个 1

N=12, 写下 1,2,3,4,5, 6,7, 8,9,10,11,12, 这样 1 的个数是 5

问题是：写一个函数 f(N)，返回 1 到 N 之间出现的 1 的个数，比如 f(12) = 5

核心 规律总结

```

long long count1_divide(int n)
{
    long long lCount = 0;
    long long lHigh, lLow, lCur;
    lHigh = lLow = lCur = 0;
    long long lFactor = 1;
    while(n / lFactor)
    {
        lLow = n - (n/lFactor)*lFactor; //参见123 - (123/10)*10
        lCur = (n / lFactor) % 10; //参见(123/10)%10
        lHigh = n / (lFactor * 10); //参见123 / (10*10)
        switch(lCur)
        {
            case 0: lCount += lHigh * lFactor; //如果当前位为0，那么当前位出现1的次数 = 高位*位数
                    break;
            case 1: lCount += lHigh * lFactor + lLow + 1; //如果当前位为1，那么当前位出现1的次数 = 高位 * 位数 + 低位 + 1
                    break;
            default: lCount += (lHigh + 1) * lFactor; //如果当前位>=2，那么当前为出现1的次数 = (高位+1)*位数
        }
        lFactor *= 10;
    }
    return lCount;
}

```

3.1.5 寻找最大的 k 个数

3.1.6 精确表达浮点数

在计算机中，使用 float 或 double 来存储小数不能得到精确值。希望得到精确值，最好用分数形式来表示小数。有限小数或者无限循环小数可以转化为分数

例如：

```
0.9 = 9/10
```

```
0.333(3) = 1/3(括号中的数字表示是循环节)
```

```
//一个小数可以用好几种分数形式来表示。如：
```

```
0.333(3) = 1/3 = 3/9
```

```
//给定一个有限小数或者无限循环小数，你能否用分母最小的分数形式来返回这个小数呢？如果输入为  
循环小数，循环节用括号标记出来。下面是一些可能的输入数据，  
如0.3,0.30,0.3(000),0.3333(3333)、.....
```

3.1.7 最大公约数问题

3.1.8 找到符合条件的整数

3.1.9 斐波那契数列

3.1.10 寻找数组中的最大最小值

3.1.11 寻找最近点对

3.1.12 找到符合条件的两个数

3.1.13 子数组的最大乘积

给定一个长度为 N 的整数数组，只允许用乘法，不能用除法，计算任意 (N-1) 个数的组合中乘积最大的一组，并写出算法的时间复杂度

3.1.14 子数组之和的最大值

3.1.15 子数组之和的最大值 (二维)

3.1.16 求数组中最长递增子序列

写一个时间复杂度尽可能低的程序，求一个一维数组 (N 个元素) 中最长递增子序列的长度。

例如，在序列 1,-1,2,-3,4,-5,6,-7 中，最长递增子序列的长度为 4(如 1, 2,4,6)

3.1.17 数组循环移位

3.1.18 数组分割

3.1.19 区间重合判断

给定一个源区间 $[x,y]$ ($y \geq x$) 和 N 个无序的目标区间 $[x_1,y_1],[x_2,y_2],[x_3,y_3],\dots,[x_n,y_n]$, 判断源区间 $[x,y]$ 是不是在目标区间内 (也即 $[x,y]$ 是否属于任意 $[x_i,y_i]$)

例如: 给定源区间 $[1,6]$ 和一组无序的目标区间 $[2,3][1,2][3,9]$, 即可认为区间 $[1,6]$ 在区间 $[2,3][1,2][3,9]$ 内 (因为目标区间实际上是 $[1,9]$)

3.1.20 程序理解和时间分析

3.1.21 只考加法的面试题

3.2 字符串及链表的探索

3.2.1 字符串移位包含的问题

3.2.2 电话号码对应英语单词

3.2.3 计算字符串的相似度

3.2.4 从无头单链表中删除节点

3.2.5 最短摘要的生成

3.2.6 判断两个链表是否交叉

3.2.7 队列中取最大值操作问题

3.2.8 求二叉树中结点的最大距离

3.2.9 重建二叉树

输入某二叉树的前序遍历和中序遍历, 请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含有重复的数字。

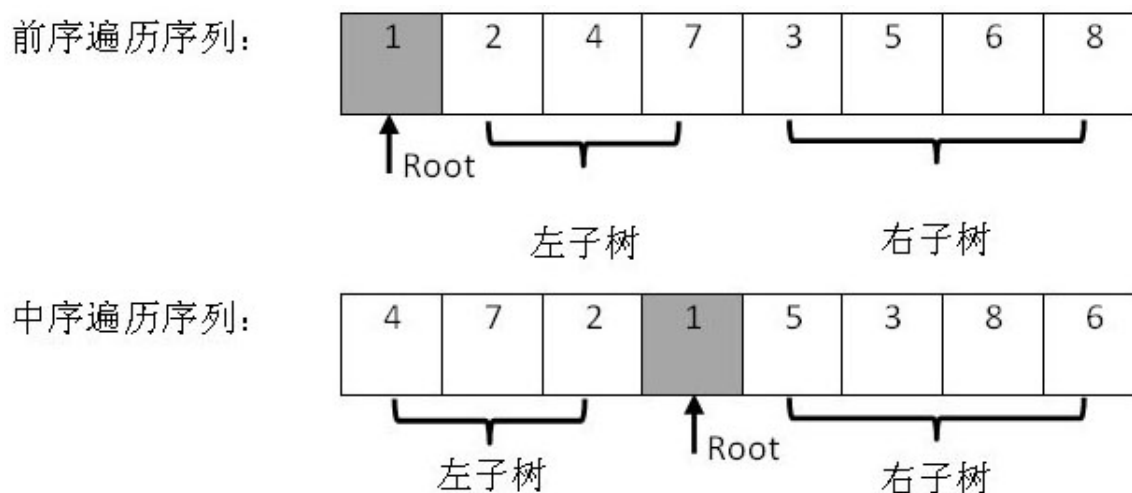
例如, 前序遍历序列: 1, 2, 3, 7, 3, 5, 6, 8, 中序遍历序列: 4, 7, 2, 1, 5, 3, 8, 6

分析

在二叉树的前序遍历序列中, 第一个数字总是树的根结点的值。但在中序遍历序列中, 根结点的值在序列的中间, 左子树的结点的值位于根结点的值的左边, 而右子树的结点的值位于根结点的值的右边。因此我们需要扫描中序遍历序列, 才能找到根结点的值。

前序遍历序列的第一个数字 1 就是根结点的值。扫描中序遍历序列，就能确定根结点的值的位置。根据中序遍历特点，在根结点的值 1 前面的 3 个数字都是左子树结点的值，位于 1 后面的数字都是右子树结点的值。

在二叉树的前序遍历和中序遍历的序列中确定根结点的值、左子树结点的值和右子树结点的值的步骤如下图所示：



分别找到了左、右子树的前序遍历序列和中序遍历序列，我们就可以用同样的方法分别去构建左右子树。换句话说，这是一个递归的过程。

思路总结：先根据前序遍历序列的第一个数字创建根结点，接下来在中序遍历序列中找到根结点的位置，这样就能确定左、右子树结点的数量。在前序遍历和中序遍历的序列中划分了左、右子树结点的值之后，就可以递归地去分别构建它的左右子树。

要点

- 边界条件 `preOrder` 与 `inOrder` 是否为空
- 终止条件 `leftLength > 0` 与 `rightLength > 0` 时才重构子树
- 分解子树

```
public static Node<int> Construct(int[] preOrder, int[] inOrder, int length)
{
    // 空指针判断
    if (preOrder == null || inOrder == null || length <= 0)
    {
        return null;
    }

    return ConstructCore(preOrder, 0, preOrder.Length - 1, inOrder, 0, inOrder.Length
        - 1);
}
```

```

public static Node<int> ConstructCore(int[] preOrder, int startPreOrder, int
    endPreOrder, int[] inOrder, int startInOrder, int endInOrder)
{
    // 前序遍历序列的第一个数字是根结点的值
    int rootValue = preOrder[startPreOrder];
    Node<int> root = new Node<int>();
    root.data = rootValue;
    root.lchild = root.rchild = null;

    if (startPreOrder == endPreOrder)
    {
        if (startInOrder == endInOrder &&
            preOrder[startPreOrder] == inOrder[startInOrder])
        {
            return root;
        }
        else
        {
            throw new Exception("Invalid_input!");
        }
    }

    // 在中序遍历中找到根结点的值
    int rootInOrder = startInOrder;
    while (rootInOrder <= endInOrder && inOrder[rootInOrder] != rootValue)
    {
        rootInOrder++;
    }

    // 输入的两个序列不匹配的情况
    if (rootInOrder == endInOrder && inOrder[rootInOrder] != rootValue)
    {
        throw new Exception("Invalid_input!");
    }

    int leftLength = rootInOrder - startInOrder;
    int leftPreOrderEnd = startPreOrder + leftLength;
    if (leftLength > 0)
    {
        // 构建左子树
        root.lchild = ConstructCore(preOrder, startPreOrder + 1, leftPreOrderEnd,
            inOrder, startInOrder, rootInOrder - 1);
    }
    if (leftLength < endPreOrder - startPreOrder)
    {
        // 构建右子树
        root.rchild = ConstructCore(preOrder, leftPreOrderEnd + 1, endPreOrder,

```

```

        inOrder, rootInOrder + 1, endInOrder);
    }

    return root;
}

```

3.2.10 分层遍历二叉树

- 递归
- 如何体现层次化

```

void printLevelNode_index(Node* pRoot)
{
    int iCur = 0, iLast = 1; //初始化游标起始位置和当前层结束位置的下一个位置
    vector<Node*> vecNode;
    vecNode.push_back(pRoot);
    while(iCur < vecNode.size()) //这里应该是层结束位置等于数组大小的时候结束循环
    {
        iLast = vecNode.size(); //牛逼，直接用当前层结束位置的下一位置 等于 向量大小 来解决
                                //了每次的更新层结束位置的问题
        while(iCur < iLast)
        {
            Node* pNode = vecNode[iCur];
            printf("%d ", pNode->_iVal); //应该先打印出当前节点值
            if(pNode->_pLeft)
            {
                vecNode.push_back(pNode->_pLeft);
            }
            if(pNode->_pRight)
            {
                vecNode.push_back(pNode->_pRight);
            }
            iCur++;
        }
        printf("\n");
    }
}

```

3.2.11 二分查找

核心

- 防止溢出 `mid = minIndex + (maxIndex - minIndex)/2`
- 结束条件 `while(left <= right)`

3.3 数学游戏

3.3.1 金刚坐飞机问题

3.3.2 瓷砖覆盖地板

3.3.3 买票找零

3.3.4 点是否在三角形内

3.3.5 磁带文件存放优化

3.3.6 桶中取黑白球

3.3.7 蚂蚁爬杆

3.3.8 三角形测试用例

3.3.9 数独知多少

3.3.10 数字哑谜和回文

3.3.11 扫雷游戏的概率

第四章 Array- 80

4.1 Two Sum

4.1.1 问题

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target. You may assume that each input would have exactly one solution.

Example Given nums = [2, 7, 11, 15], target = 9,
Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].

4.1.2 解答

4.1.3 Normal-两层循环

4.1.4 Good-HashTable:HashMap

- **Time complexity** : $O(n)$. We traverse the list containing n elements only once. Each look up in the table costs only $O(1)$ time.
- **Space complexity** : $O(n)$. The extra space required depends on the number of items stored in the hash table, which stores at most n elements.

```
//\////////////////////////////////Two-pass Hash Table////////////////////////////////\
#include <unordered_map>
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        vector<int> result;
        unordered_multimap<int, int> intergers;
        for(int i = 0; i < nums.size(); ++i)
        {
            // map 的插入值操作
            intergers.insert(pair<int,int>(nums[i],i));
        }
    }
};
```


4.2.1 分析

4.2.2 解答

4.3 Word Ladder II

Given two words (beginWord and endWord), and a dictionary's word list, find all shortest transformation sequence(s) from beginWord to endWord, such that:

1-Only one letter can be changed at a time

2-Each transformed word must exist in the word list. Note that beginWord is not a transformed word.

Note:

Return an empty list if there is no such transformation sequence.

All words have the same length.

All words contain only lowercase alphabetic characters.

You may assume no duplicates in the word list.

You may assume beginWord and endWord are non-empty and are not the same.

Examples Given:

beginWord = "hit"

endWord = "cog"

wordList = ["hot", "dot", "dog", "lot", "log", "cog"]

Return

```
[
  ["hit", "hot", "dot", "dog", "cog"],
  ["hit", "hot", "lot", "log", "cog"]
]
```

4.3.1 分析

4.3.2 解答

<http://www.cnblogs.com/ShaneZhang/p/3748494.html>

4.4 4Sum

Given an array S of n integers, are there elements a, b, c, and d in S such that $a + b + c + d = \text{target}$? Find all unique quadruplets in the array which gives the sum of target.

Note: The solution set must not contain duplicate quadruplets.

Examples given array $S = [1, 0, -1, 0, -2, 2]$, and $\text{target} = 0$.

A solution set is:

```
[  
  [-1, 0, 0, 1],  
  [-2, -1, 1, 2],  
  [-2, 0, 0, 2]  
]
```

4.4.1 分析

4.4.2 解答

4.5 Rotate Array

Rotate an array of n elements to the right by k steps.

Note: Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.

Examples with $n = 7$ and $k = 3$, the array $[1, 2, 3, 4, 5, 6, 7]$ is rotated to $[5, 6, 7, 1, 2, 3, 4]$.

4.5.1 分析

4.5.2 解答

4.6 Maximum Product Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

Examples Given the array $[2, 3, -2, 4]$,
the contiguous subarray $[2, 3]$ has the largest product $= 6$.

4.6.1 分析

4.6.2 解答

4.7 Spiral Matrix

Given a matrix of $m \times n$ elements (m rows, n columns), return all elements of the matrix in spiral order.

Examples Given the following matrix:

```
[  
[ 1, 2, 3 ],  
[ 4, 5, 6 ],  
[ 7, 8, 9 ]  
]
```

You should return [1,2,3,6,9,8,7,4,5].

4.7.1 分析

4.7.2 解答

4.8 First Missing Positive

Given an unsorted integer array, find the first missing positive integer.

Examples Given [1,2,0] return 3,
and [3,4,-1,1] return 2.

Your algorithm should run in $O(n)$ time and uses constant space.

4.8.1 分析

4.8.2 解答

4.9 Word Search

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

Examples Given board =

```
[  
['A','B','C','E'],  
['S','F','C','S'],  
['A','D','E','E']  
]
```

word = "ABCCED", -> returns true,

word = "SEE", -> returns true,

word = "ABCB", -> returns false.

4.9.1 分析

4.9.2 解答

4.10 Largest Rectangle in Histogram

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

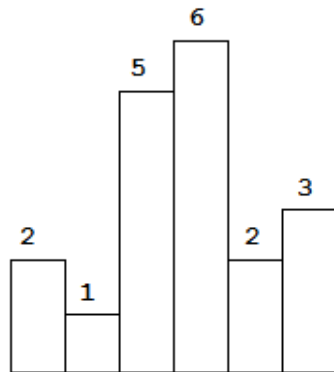


Fig 4.1: Above is a histogram where width of each bar is 1, given height = $[2,1,5,6,2,3]$

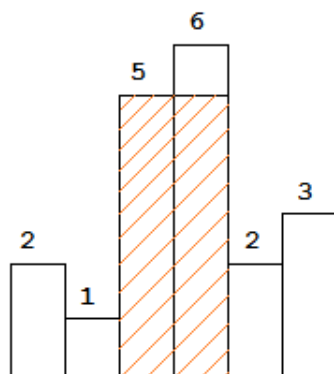


Fig 4.2: The largest rectangle is shown in the shaded area, which has area = 10 unit.

Examples Given heights = $[2,1,5,6,2,3]$,
return 10.

4.10.1 分析

4.10.2 解答

4.11 Jump Game II

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.
Your goal is to reach the last index in the minimum number of jumps.

Examples Given array $A = [2,3,1,1,4]$

The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

Note: You can assume that you can always reach the last index.

4.11.1 分析

4.11.2 解答

4.12 Maximal Rectangle

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

Examples given the following matrix:

1 0 1 0 0

1 0 1 1 1

1 1 1 1 1

1 0 0 1 0

Return 6.

4.12.1 分析

4.12.2 解答

第五章 Hash Table- 52

5.1 Longest Substring Without Repeating Characters

问题

Given a string, find the length of the longest substring without repeating characters.

Examples Given "abcabcbb", the answer is "abc", which the length is 3.

Given "bbbb", the answer is "b", with the length of 1.

Given "pwwkew", the answer is "wke", with the length of 3. Note that the answer must be a substring, "pwke" is a subsequence and not a substring.

分析

如果发现重复：第一个指针指向以查找的序列的最后一个该字符，另一个指向当前的，求取最长的中间长度。

解答

```
class Solution {
public:
    int lengthOfLongestSubstring(string s)
    {
        int maxLen = 0, repeatCharacter_First_index = -1;
        unordered_multimap<char, int> subString;
        for (int i = 0; i < s.length(); ++i)
        {
            unordered_multimap<char, int>::const_iterator it = subString.find(s[i]);
            if (it != subString.end())
            {
                int max_second = it->second;
                auto range = subString.equal_range(s[i]);
                for_each(range.first, range.second, [&max_second](unordered_multimap<char,
                    int>::value_type& x){max_second = max(max_second, x.second); });
                repeatCharacter_First_index = max(repeatCharacter_First_index, max_second);
                // [...First_max]...Second_not insert
            }
        }
    }
};
```

```
        subString.insert(pair<char, int>(s[i], i));
        maxlen = max(maxLen, i - repeatCharacter_First_index );
    }
    return maxLen;
}
};
```

5.2

Examples

5.2.1 分析

5.2.2 解答

5.3

Examples

5.3.1 分析

5.3.2 解答

5.4

Examples

5.4.1 分析

5.4.2 解答

5.5

Examples

5.5.1 分析

5.5.2 解答

5.6

Examples

5.6.1 分析

5.6.2 解答

5.7

Examples

5.7.1 分析

5.7.2 解答

5.8

Examples

5.8.1 分析

5.8.2 解答

5.9

Examples

5.9.1 分析

5.9.2 解答

第六章 Linked List- 27

<http://blog.csdn.net/luckyxiaoqiang/article/details/7393134/>

6.1 将单链表反转

6.1.1 解答

6.2 查找单链表中的倒数第 K 个结点 ($k > 0$)

双指针

6.2.1 解答

6.3 查找单链表的中间结点

双指针

6.3.1 解答

6.4 从尾到头打印单链表

Examples

6.4.1 分析

6.4.2 解答

6.5 已知两个单链表 pHead1 和 pHead2 各自有序，把它们合并成一个链表依然有序

Examples

6.5.1 分析

6.5.2 解答

6.6 判断一个单链表中是否有环

Examples

6.6.1 分析

6.6.2 解答

6.7 判断两个单链表是否相交

6.7.1 分析

<http://blog.csdn.net/jiqiren007/article/details/6572685>

方法 1. 将第一个链表的尾部指向第二个链表，如果相交则必产生环，且环的头结点为第二个链表的头结点，没有相交则不存在环。

方法 2. 如果两个链表相交，则相交后的节点都一样，所以结尾节点也将一样，我们以此遍历第一个第二个链表，取出其尾部地址，判断是否相等即可。

6.7.2 解答

6.8 求两个单链表相交的第一个节点

Examples

6.8.1 分析

6.8.2 解答

6.9 已知一个单链表中存在环，求进入环中的第一个节点

Examples

6.9.1 分析

6.9.2 解答

6.10 给出一单链表头指针 pHead 和一节点指针 pToBeDeleted, O(1) 时间复杂度删除节点 pToBeDeleted

6.10.1 分析

对于删除节点，我们普通的思路就是让该节点的前一个节点指向该节点的下一个节点，这种情况需要遍历找到该节点的前一个节点，时间复杂度为 $O(n)$ 。对于链表，链表中的每个节点结构都是一样的，所以我们可以把该节点的下一个节点的数据复制到该节点，然后删除下一个节点即可。要注意最后一个节点的情况，这个时候只能用常见的方法来操作，先找到前一个节点，但总体的平均时间复杂度还是 $O(1)$ 。

6.10.2 解答

第七章 Math- 58

7.1

Examples

7.1.1 分析

7.1.2 解答

7.2

Examples

7.2.1 分析

7.2.2 解答

7.3

Examples

7.3.1 分析

7.3.2 解答

7.4

Examples

7.4.1 分析

7.4.2 解答

7.5

Examples

7.5.1 分析

7.5.2 解答

7.6

Examples

7.6.1 分析

7.6.2 解答

7.7

Examples

7.7.1 分析

7.7.2 解答

7.8

Examples

7.8.1 分析

7.8.2 解答

7.9

Examples

7.9.1 分析

7.9.2 解答

7.10

Examples

7.10.1 分析

7.10.2 解答

第八章 Two Pointers- 33

8.1

Examples

8.1.1 分析

8.1.2 解答

8.2

Examples

8.2.1 分析

8.2.2 解答

8.3

Examples

8.3.1 分析

8.3.2 解答

8.4

Examples

8.4.1 分析

8.4.2 解答

8.5

Examples

8.5.1 分析

8.5.2 解答

8.6

Examples

8.6.1 分析

8.6.2 解答

8.7

Examples

8.7.1 分析

8.7.2 解答

8.8

Examples

8.8.1 分析

8.8.2 解答

8.9

Examples

8.9.1 分析

8.9.2 解答

8.10

Examples

8.10.1 分析

8.10.2 解答

第九章 String- 55

9.1

Examples

9.1.1 分析

9.1.2 解答

9.2

Examples

9.2.1 分析

9.2.2 解答

9.3

Examples

9.3.1 分析

9.3.2 解答

9.4

Examples

9.4.1 分析

9.4.2 解答

9.5

Examples

9.5.1 分析

9.5.2 解答

9.6

Examples

9.6.1 分析

9.6.2 解答

9.7

Examples

9.7.1 分析

9.7.2 解答

9.8

Examples

9.8.1 分析

9.8.2 解答

9.9

Examples

9.9.1 分析

9.9.2 解答

9.10

Examples

9.10.1 分析

9.10.2 解答

9.11

Examples

9.11.1 分析

9.11.2 解答

9.12

Examples

9.12.1 分析

9.12.2 解答

9.13

Examples

9.13.1 分析

9.13.2 解答

9.14

Examples

9.14.1 分析

9.14.2 解答

9.15

Examples

9.15.1 分析

9.15.2 解答

9.16 Divide and Conquer- 12

9.17

Examples

9.17.1 分析

9.17.2 解答

9.18

Examples

9.18.1 分析

9.18.2 解答

9.19

Examples

9.19.1 分析

9.19.2 解答

9.20

Examples

9.20.1 分析

9.20.2 解答

9.21

Examples

9.21.1 分析

9.21.2 解答

9.22

Examples

9.22.1 分析

9.22.2 解答

9.23

Examples

9.23.1 分析

9.23.2 解答

9.24

Examples

9.24.1 分析

9.24.2 解答

9.25

Examples

9.25.1 分析

9.25.2 解答

9.26

Examples

9.26.1 分析

9.26.2 解答

第十章 Binary Search- 38

10.1

Examples

10.1.1 分析

10.1.2 解答

10.2

Examples

10.2.1 分析

10.2.2 解答

10.3

Examples

10.3.1 分析

10.3.2 解答

10.4

Examples

10.4.1 分析

10.4.2 解答

10.5

Examples

10.5.1 分析

10.5.2 解答

10.6

Examples

10.6.1 分析

10.6.2 解答

10.7

Examples

10.7.1 分析

10.7.2 解答

10.8

Examples

10.8.1 分析

10.8.2 解答

10.9

Examples

10.9.1 分析

10.9.2 解答

10.10

Examples

10.10.1 分析

10.10.2 解答

第十一章 Dynamic Programming- 66

11.1

Examples

11.1.1 分析

11.1.2 解答

11.2

Examples

11.2.1 分析

11.2.2 解答

11.3

Examples

11.3.1 分析

11.3.2 解答

11.4

Examples

11.4.1 分析

11.4.2 解答

11.5

Examples

11.5.1 分析

11.5.2 解答

11.6

Examples

11.6.1 分析

11.6.2 解答

11.7

Examples

11.7.1 分析

11.7.2 解答

11.8

Examples

11.8.1 分析

11.8.2 解答

11.9

Examples

11.9.1 分析

11.9.2 解答

11.10

Examples

11.10.1 分析

11.10.2 解答

11.11

Examples

11.11.1 分析

11.11.2 解答

11.12

Examples

11.12.1 分析

11.12.2 解答

11.13

Examples

11.13.1 分析

11.13.2 解答

11.14

Examples

11.14.1 分析

11.14.2 解答

11.15

Examples

11.15.1 分析

11.15.2 解答

11.16

Examples

11.16.1 分析

11.16.2 解答

11.17

Examples

11.17.1 分析

11.17.2 解答

11.18

Examples

11.18.1 分析

11.18.2 解答

第十二章 Backtracking- 34

12.1

Examples

12.1.1 分析

12.1.2 解答

12.2

Examples

12.2.1 分析

12.2.2 解答

12.3

Examples

12.3.1 分析

12.3.2 解答

12.4

Examples

12.4.1 分析

12.4.2 解答

12.5

Examples

12.5.1 分析

12.5.2 解答

12.6

Examples

12.6.1 分析

12.6.2 解答

12.7

Examples

12.7.1 分析

12.7.2 解答

第十三章 Stack- 25

13.1

Examples

13.1.1 分析

13.1.2 解答

13.2

Examples

13.2.1 分析

13.2.2 解答

13.3

Examples

13.3.1 分析

13.3.2 解答

13.4

Examples

13.4.1 分析

13.4.2 解答

13.5

Examples

13.5.1 分析

13.5.2 解答

第十四章 Heap- 15

14.1

Examples

14.1.1 分析

14.1.2 解答

14.2

Examples

14.2.1 分析

14.2.2 解答

14.3

Examples

14.3.1 分析

14.3.2 解答

14.4

Examples

14.4.1 分析

14.4.2 解答

14.5

Examples

14.5.1 分析

14.5.2 解答

14.6

Examples

14.6.1 分析

14.6.2 解答

14.7

Examples

14.7.1 分析

14.7.2 解答

14.8

Examples

14.8.1 分析

14.8.2 解答

14.9

Examples

14.9.1 分析

14.9.2 解答

14.10

Examples

14.10.1 分析

14.10.2 解答

第十五章 Greedy- 19

15.1

Examples

15.1.1 分析

15.1.2 解答

15.2

Examples

15.2.1 分析

15.2.2 解答

15.3

Examples

15.3.1 分析

15.3.2 解答

15.4

Examples

15.4.1 分析

15.4.2 解答

15.5

Examples

15.5.1 分析

15.5.2 解答

15.6

Examples

15.6.1 分析

15.6.2 解答

15.7

Examples

15.7.1 分析

15.7.2 解答

15.8

Examples

15.8.1 分析

15.8.2 解答

15.9

Examples

15.9.1 分析

15.9.2 解答

15.10

Examples

15.10.1 分析

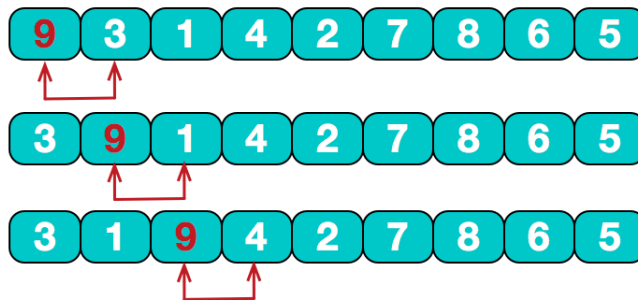
15.10.2 解答

第十六章 Sort- 16

16.1 冒泡

冒泡排序的基本思想是，对相邻的元素进行两两比较，顺序相反则进行交换，这样，每一趟会将最小或最大的元素“浮”到顶端，最终达到完全有序

相邻元素两两比较，反序则交换



第一轮完毕，将最大元素9浮到数组顶端



同理,第二轮将第二大元素8浮到数组顶端



排序完成



16.1.1 解答

<https://github.com/ctzhenghua/C-NetworkPractice-Code/blob/master/DataStructure/Sort/PopSort.cc>

16.2 选择排序

选择排序的思想非常直接，不是要排序么？那好，我就从所有序列中先找到最小的，然后放到第一个位置。之后再看剩余元素中最小的，放到第二个位置……以此类推，就可以完成整个的排序工作了。可以很清楚的发现，选择排序是固定位置，找元素。相比于插入排序的固定元素找位置，是两种思维方式。不过条条大路通罗马，两者的目的是一样的。

16.2.1 解答

<https://github.com/ctzhenghua/C-NetworkPractice-Code/blob/master/DataStructure/Sort/SelectSort.cc>

16.3 快速排序

快速排序是找出一个元素（理论上可以随便找一个）作为基准 (pivot), 然后对数组进行分区操作, 使基准左边元素的值都不大于基准值, 基准右边的元素值都不小于基准值, 如此作为基准的元素调整到排序后的正确位置。递归快速排序, 将其他 $n-1$ 个元素也调整到排序后的正确位置。最后每个元素都是在排序后的正确位置, 排序完成。所以快速排序算法的核心算法是分区操作, 即如何调整基准的位置以及调整返回基准的最终位置以便分治递归。

16.3.1 解答

<https://github.com/ctzhenghua/C-NetworkPractice-Code/blob/master/DataStructure/Sort/QuickSort.cc>

16.4 插入排序

插入即表示将一个新的数据插入到一个有序数组中, 并继续保持有序。例如有一个长度为 N 的无序数组, 进行 $N-1$ 次的插入即能完成排序; 第一次, 数组第 1 个数认为是有序的数组, 将数组第二个元素插入仅有 1 个有序的数组中; 第二次, 数组前两个元素组成有序的数组, 将数组第三个元素插入由两个元素构成的有序数组中……第 $N-1$ 次, 数组前 $N-1$ 个元素组成有序的数组, 将数组的第 N 个元素插入由 $N-1$ 个元素构成的有序数组中, 则完成了整个插入排序。

16.4.1 解答

16.5 归并

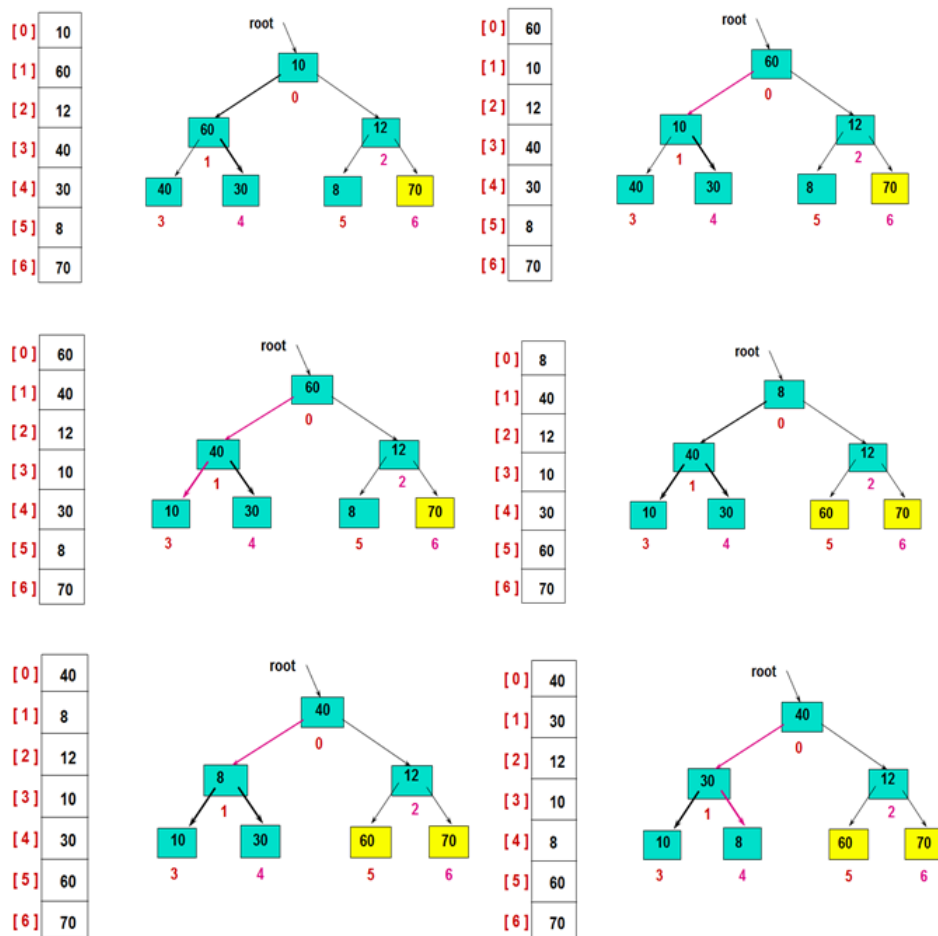
<http://www.cnblogs.com/jingmoxukong/p/4308823.html>

16.5.1 解答

16.6 堆

<http://www.cnblogs.com/mengdd/archive/2012/11/30/2796845.html>

- 数组最后总是不需要再调整的
- 调整后，根与当前剩下未确定的最后一个进行位置交换



16.6.1 解答

<http://blog.csdn.net/xiaoxiaoxuewen/article/details/7570621/>

16.7

Examples

16.7.1 分析

16.7.2 解答

16.8

Examples

16.8.1 分析

16.8.2 解答

16.9

Examples

16.9.1 分析

16.9.2 解答

16.10

Examples

16.10.1 分析

16.10.2 解答

第十七章 Bit Manipulation- 26

17.1

Examples

17.1.1 分析

17.1.2 解答

17.2

Examples

17.2.1 分析

17.2.2 解答

17.3

Examples

17.3.1 分析

17.3.2 解答

17.4

Examples

17.4.1 分析

17.4.2 解答

17.5

Examples

17.5.1 分析

17.5.2 解答

17.6

Examples

17.6.1 分析

17.6.2 解答

17.7

Examples

17.7.1 分析

17.7.2 解答

17.8

Examples

17.8.1 分析

17.8.2 解答

17.9

Examples

17.9.1 分析

17.9.2 解答

17.10

Examples

17.10.1 分析

17.10.2 解答

第十八章 Tree- 48

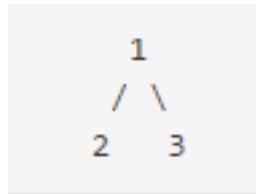
<http://blog.csdn.net/luckyxiaoqiang/article/details/7518888>

18.1 Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain at least one node and does not need to go through the root.

Examples Given the below binary tree,



Return 6.

18.1.1 分析

18.1.2 解答

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    int maxPathSum(TreeNode* root) {
```

```
}  
};
```

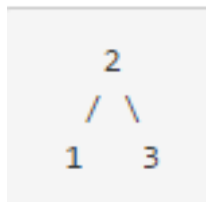
18.2 Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

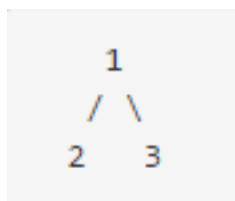
Assume a BST is defined as follows:

- 1-The left subtree of a node contains only nodes with keys less than the node's key.
- 2-The right subtree of a node contains only nodes with keys greater than the node's key.
- 3-Both the left and right subtrees must also be binary search trees.

Examples Given the below binary tree,



Binary tree [2,1,3], return true.



Binary tree [1,2,3], return false.

18.2.1 分析

18.2.2 解答

18.3 求二叉树中的节点个数

18.3.1 分析

18.3.2 解答

18.4 求二叉树的深度

18.4.1 分析

18.4.2 解答

18.5 前序遍历，中序遍历，后序遍历

18.5.1 分析

18.5.2 解答

18.6 分层遍历二叉树（按层次从上往下，从左往右）

18.6.1 分析

18.6.2 解答

18.7 将二叉查找树变为有序的双向链表

18.7.1 分析

18.7.2 解答

18.8 求二叉树第 K 层的节点个数

18.8.1 分析

18.8.2 解答

18.9 求二叉树中叶子节点的个数

18.9.1 分析

18.9.2 解答

第十九章 Depth-first Search- 41

19.1 部分和问题

给定整数 a_1, a_2, \dots, a_n , 判断是否可以从中选出若干数, 使他们的和恰好为 k

Examples

```
// Test Case 1
输入
n = 4
a = {1, 2, 4, 7}
k = 13

输出
Yes{13 = 2 + 4 + 7}

// Test Case 2
输入
n = 4
a = {1, 2, 4, 7}
k = 15

输出
No
```

19.1.1 分析

当 $n < 20$ 时, 运算次数为 100 0000 次, 可以接受

19.1.2 解答

```
int a[MAX_N];
int n, k;

bool dfs(int i, sum)
{
    if(i == n) return sum == k;
    if(dfs(i+1, sum)) return true;
    if(dfs(i+1, sum+a[i])) return true;
```

```
        return false;  
    }
```

19.2

Examples

19.2.1 分析

19.2.2 解答

19.3

Examples

19.3.1 分析

19.3.2 解答

19.4

Examples

19.4.1 分析

19.4.2 解答

19.5

Examples

19.5.1 分析

19.5.2 解答

19.6

Examples

19.6.1 分析

19.6.2 解答

19.7

Examples

19.7.1 分析

19.7.2 解答

19.8

Examples

19.8.1 分析

19.8.2 解答

19.9

Examples

19.9.1 分析

19.9.2 解答

19.10

Examples

19.10.1 分析

19.10.2 解答

第二十章 Breadth-first Search- 21

20.1

Examples

20.1.1 分析

20.1.2 解答

20.2

Examples

20.2.1 分析

20.2.2 解答

20.3

Examples

20.3.1 分析

20.3.2 解答

20.4

Examples

20.4.1 分析

20.4.2 解答

20.5

Examples

20.5.1 分析

20.5.2 解答

20.6

Examples

20.6.1 分析

20.6.2 解答

20.7

Examples

20.7.1 分析

20.7.2 解答

20.8

Examples

20.8.1 分析

20.8.2 解答

20.9

Examples

20.9.1 分析

20.9.2 解答

20.10

Examples

20.10.1 分析

20.10.2 解答

第二十一章 Union Find- 6

21.1

Examples

21.1.1 分析

21.1.2 解答

21.2

Examples

21.2.1 分析

21.2.2 解答

21.3

Examples

21.3.1 分析

21.3.2 解答

21.4

Examples

21.4.1 分析

21.4.2 解答

第二十二章 Graph- 10

22.1

Examples

22.1.1 分析

22.1.2 解答

22.2

Examples

22.2.1 分析

22.2.2 解答

22.3

Examples

22.3.1 分析

22.3.2 解答

22.4

Examples

22.4.1 分析

22.4.2 解答

22.5

Examples

22.5.1 分析

22.5.2 解答

22.6

Examples

22.6.1 分析

22.6.2 解答

22.7

Examples

22.7.1 分析

22.7.2 解答

22.8

Examples

22.8.1 分析

22.8.2 解答

22.9

Examples

22.9.1 分析

22.9.2 解答

22.10

Examples

22.10.1 分析

22.10.2 解答

第二十三章 Design- 27

23.1

Examples

23.1.1 分析

23.1.2 解答

23.2

Examples

23.2.1 分析

23.2.2 解答

23.3

Examples

23.3.1 分析

23.3.2 解答

23.4

Examples

23.4.1 分析

23.4.2 解答

23.5

Examples

23.5.1 分析

23.5.2 解答

23.6

Examples

23.6.1 分析

23.6.2 解答

23.7

Examples

23.7.1 分析

23.7.2 解答

23.8

Examples

23.8.1 分析

23.8.2 解答

23.9

Examples

23.9.1 分析

23.9.2 解答

23.10

Examples

23.10.1 分析

23.10.2 解答

第二十四章 Topological Sort[拓扑排序]- 5

24.1

Examples

24.1.1 分析

24.1.2 解答

24.2

Examples

24.2.1 分析

24.2.2 解答

24.3

Examples

24.3.1 分析

24.3.2 解答

第二十五章 Trie[前缀-字典树]- 7

25.1

Examples

25.1.1 分析

25.1.2 解答

25.2

Examples

25.2.1 分析

25.2.2 解答

25.3

Examples

25.3.1 分析

25.3.2 解答

第二十六章 Binary Indexed Tree- 4

26.1

Examples

26.1.1 分析

26.1.2 解答

26.2

Examples

26.2.1 分析

26.2.2 解答

26.3

Examples

26.3.1 分析

26.3.2 解答

第二十七章 Segment Tree- 4

27.1

Examples

27.1.1 分析

27.1.2 解答

27.2

Examples

27.2.1 分析

27.2.2 解答

27.3

Examples

27.3.1 分析

27.3.2 解答

第二十八章 Binary Search Tree- 4

28.1

Examples

28.1.1 分析

28.1.2 解答

28.2

Examples

28.2.1 分析

28.2.2 解答

28.3

Examples

28.3.1 分析

28.3.2 解答

第二十九章 Other Aspects

29.1 Recursion[递归]- 2

29.1.1

Examples

29.1.2 分析

29.1.3 解答

29.2 Brainteaser[谜题]- 2

29.2.1

Examples

29.2.2 分析

29.2.3 解答

29.3 Memoization[以避免递归重复计算，如 Fibonacci(斐波那契) 问题]- 1

29.3.1

Examples

29.3.2 分析

29.3.3 解答

29.4 Queue- 3

29.4.1

Examples

分析

解答

29.4.2

Examples

分析

解答

29.5 Reservoir Sampling[随机抽样]- 2

29.5.1

Examples

分析

解答

29.6 Minimax[极小化极大算法]- 3

29.6.1

Examples

分析

解答

29.6.2

Examples

分析

解答

29.7 Other Interview

29.7.1 迷宫字符串

问题

已知：一个二维的字母矩阵和一个由字母组装的单词。输出：该单词是否可以从二维字母矩阵中拼接出来，拼接规则为从矩阵的某一行的某个字母开始，持续向临近的字母扩展（向上、向下、向左或向右），直至拼接出该单词。若最终可以拼接出，则输出 true；若最终不可以拼接出，则输出 false

Example Given 字母矩阵为

A B C D E

E F G H U

A B S D F

输入：ABCGH 输出 true

输入：FGHDSB 输出 true

输入：EFGU 输出 false

解答

队列方法

```
#include <iostream>
#include <list>
using namespace std;

struct Roud{
    int i; //rows
    int j; //columns
    int now; //testCase[now];
    list<char> roud;
};

// queue
list<Roud> all;
list<char> result;

// arrayCharacters
char array[3][5] = { { 'A', 'B', 'C', 'D', 'E' }, { 'E', 'F', 'G', 'H', 'U' }, { 'A',
    'B', 'S', 'D', 'F' } };
```

```

void createReslut(char * testCase)
{
    for (int i = 0; i < strlen(testCase); ++i)
    {
        result.push_back(testCase[i]);
    }
}

void solve(char * testCase)
{
    // Checking The First Elements
    int i = 0;
    for (int _i = 0; _i < 3; ++_i)
    {
        for (int j = 0; j < 5; ++j)
        {
            if (array[_i][j] == testCase[0])
            {
                Roud temp;
                temp.roud.push_back(testCase[0]);
                temp.i = _i;
                temp.j = j;
                temp.now = 0;
                all.push_back(temp);

                if (i == 0)
                    ++i;
            }
        }
    }

    // Queue Checking Roud
    while (testCase[i] != '\0' && all.size() > 0)
    {
        Roud temp = all.front();
        Roud tempN;
        if (result == temp.roud)
            break;

        i = temp.now;
        ++i;
        // Checking 4 direction
        // Up
        if ((temp.i - 1 >= 0) && (testCase[i] == array[temp.i - 1][temp.j]))
        {
            tempN.i = temp.i - 1;
            tempN.j = temp.j;
            tempN.now = temp.now + 1;

```

```

        tempN.roud = temp.roud;
        tempN.roud.push_back(testCase[i]);
        all.push_back(tempN);
    }
    // Down
    if ((temp.i + 1 <= 2) && (testCase[i] == array[temp.i + 1][temp.j]))
    {
        tempN.i = temp.i + 1;
        tempN.j = temp.j;
        tempN.now = temp.now + 1;
        tempN.roud = temp.roud;
        tempN.roud.push_back(testCase[i]);
        all.push_back(tempN);
    }
    // Left
    if ((temp.j - 1 >= 0) && (testCase[i] == array[temp.i][temp.j-1]))
    {
        tempN.i = temp.i;
        tempN.j = temp.j-1;
        tempN.now = temp.now + 1;
        tempN.roud = temp.roud;
        tempN.roud.push_back(testCase[i]);
        all.push_back(tempN);
    }
    // Right
    if ((temp.j+1 <= 4) && (testCase[i] == array[temp.i][temp.j+1]))
    {
        tempN.i = temp.i;
        tempN.j = temp.j + 1;
        tempN.now = temp.now + 1;
        tempN.roud = temp.roud;
        tempN.roud.push_back(testCase[i]);
        all.push_back(tempN);
    }
    all.pop_front();
}

if (all.size()>0)
    cout << "true" << endl;
else
    cout << "false" << endl;
}

int main()
{
    char test[256];
    cin >> test;

```

```
    createReslut(test);  
    solve(test);  
  
    system("PAUSE");  
    return 0;  
}
```